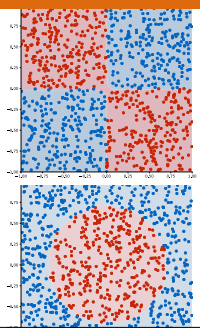
Beyond Linear Models
Part 1: Neural Networks

Machine Learning 2020
mlvu.github.io
Vrije Universiteit Amsterdam

---

making linear models more powerful

d	p	d * p
0.75	0.98	0.74
-0.66	-0.32	0.21
-0.45	0.84	-0.38
0.93	0.78	0.72
-0.42	0.24	-0.10
-0.02	0.43	-0.01
-0.74	0.58	-0.43
-0.41	-0.41	0.17
0.59	0.72	0.42



2

A few lectures ago, we saw how we could make a linear model more powerful, and able to learn nonlinear decision boundaries by just *expanding our features*: we add new features derived from the old ones, and depending on which combinations we add, we can learn new, non-linear decision boundaries or regression functions.

---

from linear to nonlinear models

neural networks specifically the feedforward network	SVMs using the kernel trick	
$\mathbf{w}^T\mathbf{x} + b$	$\mathbf{w}^T\mathbf{x} + b$	linear model
	$k(\mathbf{x}_i, \mathbf{x}_j)$	feature extractor
learns a feature extractor together with the classifier	uses a kernel to *massively* expand the feature space	

3

Both models we will see today, **neural networks** and **support vector machines**, take this idea and build on it. Neural networks are a big family, but the simplest type, the **two-layer feedforward network**, functions as a feature extractor followed by a linear model. In this case, we don't choose the extended features but we *learn* them, together with the weights of the linear model.

The support vector machine doesn't learn the expanded features (we still have to choose them manually), but it uses a **kernel function** to allow us to fit a linear model in a *very* high-dimensional feature without having to pay for actually computing all these expanded features.

---

linear models 2

1985–1995
part 1: Feedforward Neural Networks

part 2: Scalar Backpropagation

1995–2005
part 3: Support Vector Machines, Hinge loss

part 4: Optimizing under constraints
Lagrange optimization

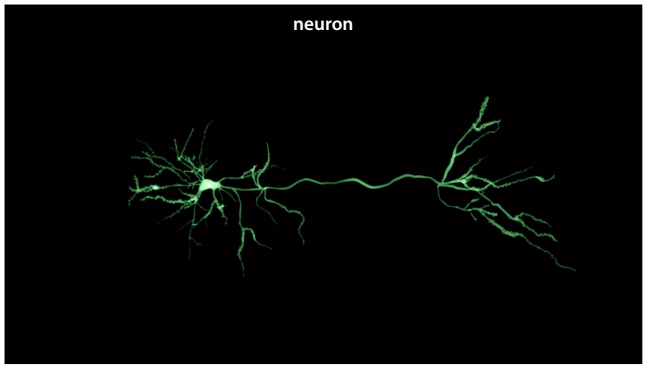
The kernel Trick

4

The layout of today's lecture will be largely chronological. We will focus on neural networks, which were very popular in the late eighties and early nineties.

Then, towards the end of the nineties, interest in neural networks died down a little and support vector machines became much more popular.

In the next lecture, we'll focus on Deep Learning, which sees neural networks make a come back in a big way.



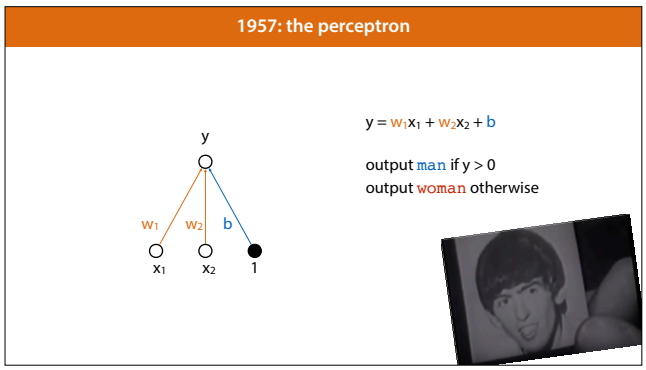
In this video, we'll start with the basics of neural networks.

In the very early days of AI (the late 1950s), researchers decided to take a simple approach to AI: the brain is the only truly intelligent system we know, so let's see what it's made of, and whether that provides some inspiration for intelligent (and learning) computer systems.

They started with a single brain cell: a neuron. A neuron receives multiple different signals from other cells through connections called **dendrites**. It processes these in a relatively simple way, deriving a single new signal, which it sends out through its single **axon**. The axon branches out so that this single output signal can reach different cells

image source: http://www.sciencealert.com/scientists-build-an-artificial-neuron-that-fully-mimics-a-human-brain-cell

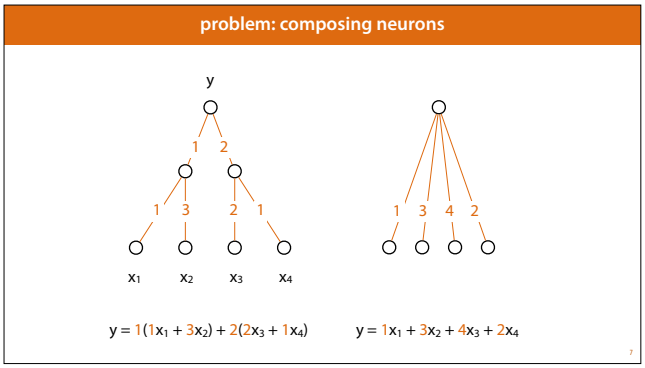
---



These ideas needed to be radically simplified to work with computers of that age, but doing so yielded one of the first successful machine learning systems: the **perceptron**. This was the model we saw in action in the video in the the first lecture.

The perceptron has a number of inputs, the *features* in modern parlance, each of which is multiplied by a weight. The result is summed, together with a bias parameter, and the sign of this result is used as the classification.

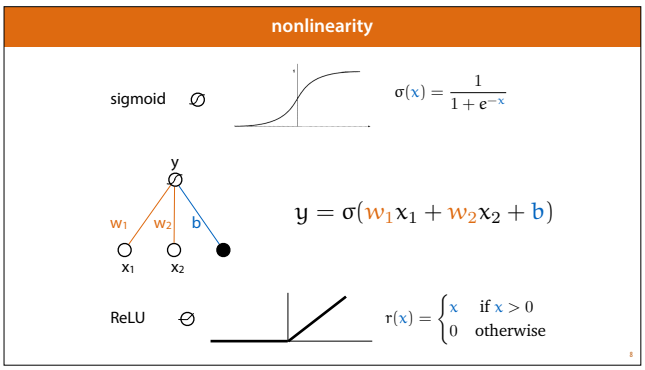
Of course, we've seen this classifier already: it's just our basic linear classifier. The training algorithm was a little different from gradient descent, but the basic principle was the same. Note that when we draw the perceptron this way, the bias can be represented as just another input that we just fix to always be 1. This is called a **bias node**.



Of course the brain's power does not come from the fact that a single neuron is such a powerful mechanism by itself: it's the *composition* of many simple parts that allows it to do what it does. We make the output of one neuron the input of another, and build networks of bllions of neurons.

And this is where the perceptron turns out to be too simple an abstraction. Because composing perceptrons doesn't make them more powerful. Consider the graph on the left, with multiple composed preceptrons.

Writing down the function that this graph represents, we see that we get a simple function, with the first two perceptrons in brackets. If we then multiply out the brackets, we see that the result is a linear function. This means that we can represent this function also as a single perceptron with four inputs. This is always true. No matter how many perceptrons you chain together, the result will never be anything more than a simple linear function over your inputs. We've removed the bias node here for simplicity, but the conclusion is the same with a bias node included.

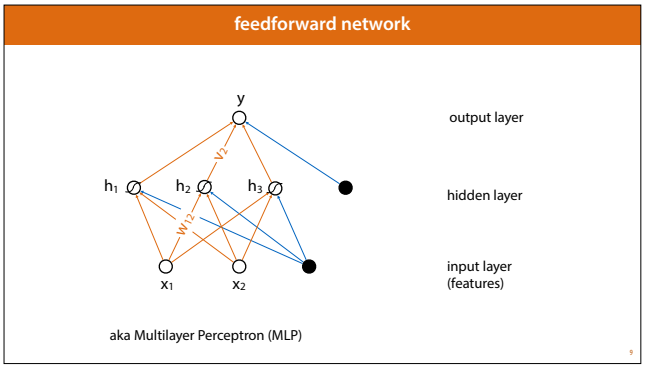


To create perceptrons that we can chain together in such a way that the result will be more expressive than any single perceptron could be, the simplest trick is to include a **non-linearity**, also called **an activation function**.

After all the weighted inputs have been combined, we pass the result through a simple non linear scalar function to produce the output. One popular option, especially in the early days of neural networks, is the **logistic sigmoid**, which we've seen already. Applying a sigmoid means that the sum of the inputs can range from negative infinity to positive infinity, but the output is always in the interval [0, 1].

Another, more recent nonlinearity is the linear rectifier, or **ReLU** nonlinearity. This function just sets every negative input to zero, and keeps everything else the same.

Not using an activation function is also called using a **linear activation**.



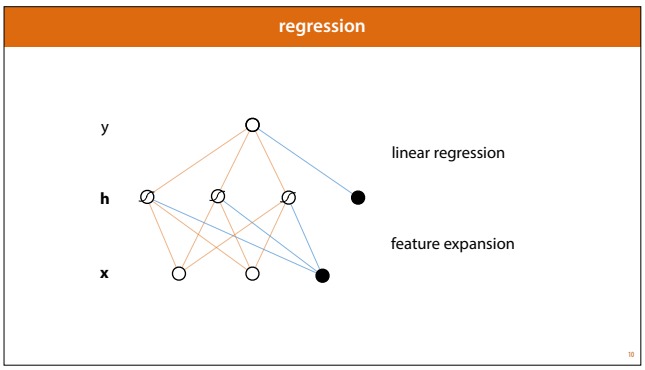
Using these nonlinearities, we can arrange single perceptrons into **neural networks**. Any arrangement of perceptrons makes a neural network, but for ease of training, this arrangement seen here was the most popular for a long time. It's called the **feedforward network** or **multilayer perceptron**. We arrange a layer of **hidden units** in the middle, each of which acts as a perceptron with a nonlinearity, connecting to all input nodes. Then we have one or more output nodes, connecting to all hidden layers. Note the following points.

- There are no cycles, the network feeds forward from input to output.
- Nodes in the same layer are not connected to each other, or to any other layer than the next and the previous one.
- Each layer is **fully connected** to the previous layer, every node in one layer connects to every node in the layer before it.

In the 80s and 90s they usually had just one hidden layer, because we hadn't figured out how to train deeper networks.

Note that Every orange and blue line in this picture represents one parameter of the model.

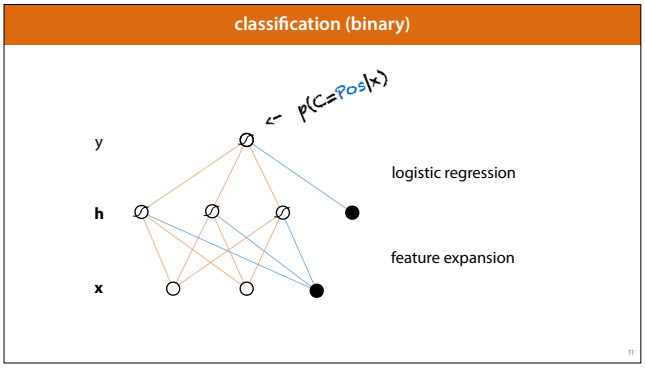
We can use networks like these to do classification or regression.



To build a regression model, all we need is one output node without an activation. This means that our network as a whole, describes a function from the feature space to the real number line.

We can think of the first layer of our network as computing a *feature expansion*: the same thing we did in the fourth lecture to enable our linear regression to learn non-linear patterns, but this time, we don't have to work out the feature expansion ourselves, we simply learn it. The second layer is then just a linear regression on this expanded feature space.

To this output, we can then apply any loss function we like, such as least-squares loss.



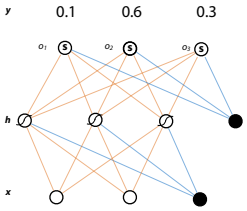
To build a classifier we can do what the perceptron did: use the sign of the output as the class. This would be like using our least squares classifier, with a feature expansion layer below it.

These days, however, it's much more common to take inspiration from the logistic regression. We apply the logistic sigmoid to the output and interpret the resulting value as the probability that the given input (x) is of the positive class.

The logarithmic loss that we saw in the last lecture, can be applied here as well.

classification (multiclass)

softmax activation



$$o_i = w^T h + b$$

$$y_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

12

For multiclass classification, we can use something called a **softmax activation**. We create a single output node for each class, and ensure that they sum to one.

The softmax function simply takes the exponent of each output node, to ensure that they are all positive, and then divides each by the sum total, to ensure that all outputs together sum to one.

After the softmax we can interpret the output of node y_3 as the probability that **x** has class 3.

Given thes probabilities, we can apply a simple log-loss: the aim is to maximize the logarithm of the probability of the true class.

---

training: *stochastic* gradient descent

pick random weights **w** (for the whole model)

loop:

  for x in X:

    w ← **w** - η∇loss$_{\mathbf{x}}$(**w**)

13

Because a neural networks can be expensive to compute we tend to use **stochastic gradient descent** to train them.

Stochastic gradient descent is very similar to the gradient descent we've seen already, but we define the loss function over a **single example** instead of summing over the whole dataset: just use the same loss function, but pretend your data set consists of only one instance.

Stochastic gradient descent has many advantages, including:

- Using a new instance each time adds some randomness to the process, which can help to escape local minima.
- Gradient descent works fine if the gradient is not perfect, but good on average (over many steps). This means that taking many small inaccurate steps is often much better than taking one very accurate big step.
- Computing the loss over the whole data is expensive. By computing loss over one instance, we get N steps of stochastic gradient descent for the price of one step of regular gradient descent.

The most common approach these days is a compromise between stochastic and regular gradient descent, where we actually compute the loss for a small **batch** of instances (say 32 of them), and take a single step of gradient descent for each batch. This is called **minibatch gradient descent,** which we'll look at more closely next lecture.

summary: training a neural network

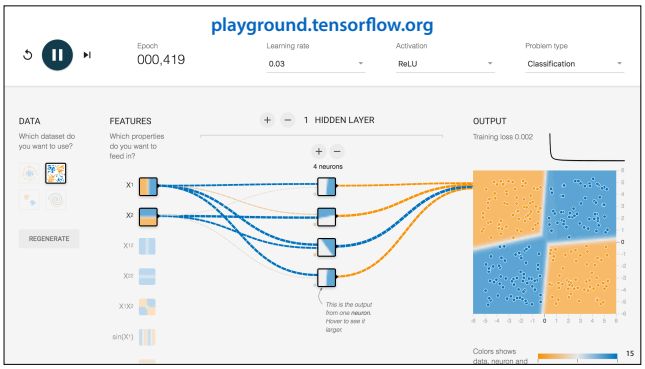
get some examples of input and output

get a loss function
least squares, cross entropy

work out the gradient of the loss wrt the weights

use (stochastic) gradient descent to improve the weights
bit by bit.

14



Before we dig into the details, we can get a sense of what this looks like in tensorflow playground.

Note:

- How the shape of the decision boundary changes based on the activation functions we choose (curvy for sigmoid, piecewise linear for ReLU)
- That adding another layer makes the network much more difficult to train (especially with sigmoid activations).

But how do we compute the gradient for such complex models?

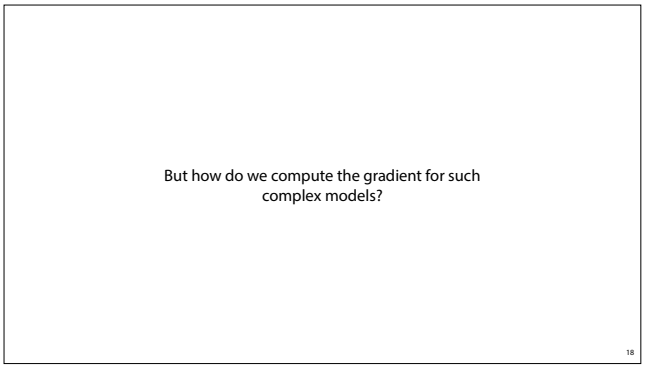
16

That's the basic idea of neural networks. So far, it's hopefully a pretty simple idea. The complexity of neural networks lies in computing the gradients. For such complex models, sitting down at the kitchen table with a stack of papers and a pencil, and working out a symbolic expression for the gradient is no longer feasible. If we manage it at all, we get horrible convoluted expressions that no longer reduce to noce, simple functions, as they did in the case of linear regression and logistic regression.

To help us out, we need the backpropagation algorithm, which we'll discuss in the next video.

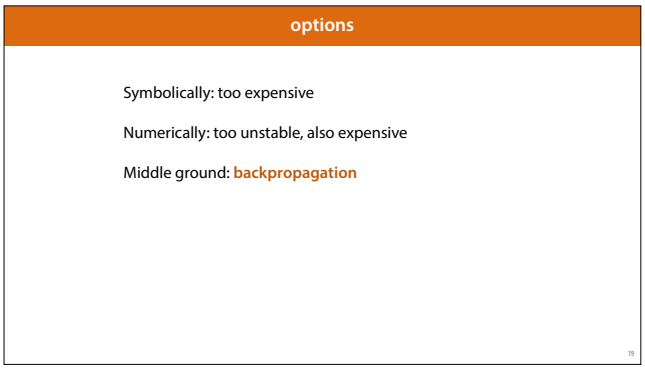
Beyond Linear Models
Part 2: Backpropagation

Machine Learning 2020
mlvu.github.io
Vrije Universiteit Amsterdam



In the last video, we saw what the structure of a very basic neural network was, and we ended on this question. How do we work out the gradient.

For neural networks, the gradients quickly get too complex to work out by hand, so we need to automate this process.



There are three basic flavors of working out derivatives and gradients automatically.

The first is to do it symbolically, What we do on pen and paper, when we work out a derivative, is a pretty mechanical process. It's not too difficult to program this process out and let the computer do it for us. This, is what happens, when you as Wolfram alpha to work out a derivative, for instance. It has its uses, cartainly, but it won;t work for us. The symbolic expression of the gradient of a function grows exponentiall with the complexity of the original function. That means that as we build bigger and bigger networks the expression of the gradient would soon grow too big to store in memory, let alone to compute.

An alternative approach is to forget the symbolic form of the function, and just estimate the gradient for a specific input $\mathbf{x}$. We could pick some points close to $\mathbf{x}$ and and fit a hyperplane through the outputs. This would be a pretty good approximation of the tangent hyperplane, so we could just read out the gradient. The problem is that this is a pretty unstable business. It's quite difficult to be sure that the answer is accurate. It's also pretty expensive: the more dimensions in your model space, the more points you need to get an accurate estimate of your gradient, and each point requires you to recompute your model for a new input.

Bakcpropagation is a middle ground: it does part of the work symbolically, and part of the work numerically. We get a very accurate computation of the gradient, and the cost of computation is usually only twice as expensive as computing the output for one input.

If we describe our model as a composition of modules, the gradient is the product of the gradient of each module with respect to its arguments.

20

This is the basic principle behind backpropagation: if we have a function, that is a composition of other functions, we can write out the derivative as repeated applications of the chain rule.

1974, 1980's: backpropagation

Break your computation down into a chain of *modules*.

Work out the derivative of each module with respect to its input symbolically.

Compute the global gradient *for a given input* by multiplying these gradients.

21

Here's the three steps required to implement backpropagation for a given function.

---

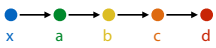
example

$$f(x) = \frac{2}{\sin(e^{-x})} \qquad f(x) = d(c(b(a(x))))$$

modules:

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^a$$
$$a(x) = -x$$

computation graph:



22

Let's show how it works on a simple example.

To show that backpropagation is a generic algorithm for working out gradients, not just a method for neural networks, we'll first show how it works for some arbitrary scalar function.

First we take our function f, and we break it up into smaller functions, the output of each feeding into the next. Defining the functions a, b, c, and d as shown, we can write f(x) = d(c(b(a(x)))).

The graph on the right is a called a **computation graph**: each node represents a small computer program that receives an input, computes an output and passes it on to another module.

Normally, we wouldn't break a function up in such small modules: this is just a simple example to illustrate the principle.

chain rule

$$\frac{\partial f(x)}{\partial x} = \frac{\partial d(c(b(a(x))))}{\partial c(b(a(x)))} \frac{\partial c(b(a(x)))}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial x}$$

$$\frac{\partial d}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

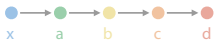
Because we've described our function as a composition of modules, we can work out the derivative purely by repeatedly applying the chain rule.

Since we know for each function what the argument is, we'll leave the arguments out to keep the notation clean.

---

example

$$f(x) = \frac{2}{\sin(e^{-x})}$$



$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^a$$
$$a(x) = -x$$

$$\boxed{\frac{\partial f}{\partial x}} = \boxed{\frac{\partial d}{\partial c}} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

global derivative

local derivatives

We'll call the derivative of the whole function the **global derivative**, and the derivative of each module with respect to its input we will call a **local derivative**.

---

work out *local* derivatives

symbolically

$$d(c) = \frac{2}{c}$$
$$c(b) = \sin b$$
$$b(a) = e^a$$
$$a(x) = -x$$

$$\frac{\partial f}{\partial x} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x}$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^a \cdot -1$$

The next step is to work out the local derivatives symbolically, using the rules we know.

The difference from what we normally do is that we stop when we have the derivatives of the output of a module in terms of the input. For instance, the derivative $\partial c / \partial b$ is $\cos b$. Normally, we would fill in the definition of b and see if we could simplify any further. Here we stop once we know the derivative in terms of b.

---

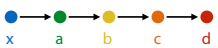
compute a forward pass (`x=-4.499`)

retain values of a, b, c, d

$$f(-4.499) = 2$$

$$d = \frac{2}{c} = 2$$
$$c = \sin b = 1$$
$$b = e^a = 90$$
$$a = -x = 4.499$$

computation graph:



Then, once all the local derivatives are known, in symbolic form, we switch to **numeric computation**. We will take a *specific* input, in this case -4.499 and compute the gradient only for that.

First we compute the output of the function given this input. This is known as a the **forward pass**. During our computation, we also retain our intermediate values a, b, c and d. These will be useful later on.

compute the backward pass

numerically

$$f(-4.499) = 2$$
$$d = \frac{2}{c} = 2$$
$$c = \sin b = 1$$
$$b = e^a = 90$$
$$a = -x = 4.499$$

$$\frac{\partial f}{\partial x} = -\frac{2}{c^2} \cdot \cos b \cdot e^a \cdot -1$$
$$= -\frac{2}{1^2} \cdot \cos 90 \cdot e^{4.499} \cdot -1$$
$$= -\frac{1}{2} \cdot 0 \cdot 90 \cdot -1 = 0$$

27

Next up is the **backward pass**. We take the chain-rule derived form of the derivative, and we fill in the intermediate values a, b, c and d.

This gives us a function with no variables, so we can compute the output. The result is that the derivative of this function, for the specific input -4.499, is 0.

---

backpropagation

Write your function as a composition of **modules**.
What the modules are is up to you.

Work out the local derivative of each module **symbolically**.

Do a **forward pass** for a given input x.
I.e. compute the function f(x), remember the intermediate values

Compute the local derivatives for x, and multiply to compute the global derivative.

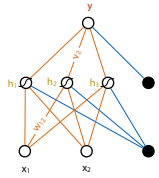
28

More fine-grained modules make the local derivatives easier to work out, but may increase the numeric instability. Less fine grained modules usually result in more accurate gradients, but if we make them too big, we end up with the problem that the symbolic expression of the gradient grows too complex.

---

backprop for a feedforward network

t: training data



$$\frac{\partial l}{\partial v_2}, \frac{\partial l}{\partial w_{12}}$$

$$l = (y - t)^2$$

$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_y$$

$$h_2 = \frac{1}{1 + \exp(-h_2')}$$

$$h_2' = w_{12} x_1 + w_{22} x_2 + b_2$$

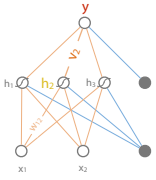
29

Let's now see how this works for a neural net.

It's important to remember that we don't care about the derivative of the output y with respect tot he inputs x. The function we're computing the gradient for is the loss, and the variables we want to compute the gradient for are the **parameters** of the network. x does end up in our computation, because it's part of the loss, but only as a constant.

---

t: training data



$$l = (y - t)^2$$
$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_h$$

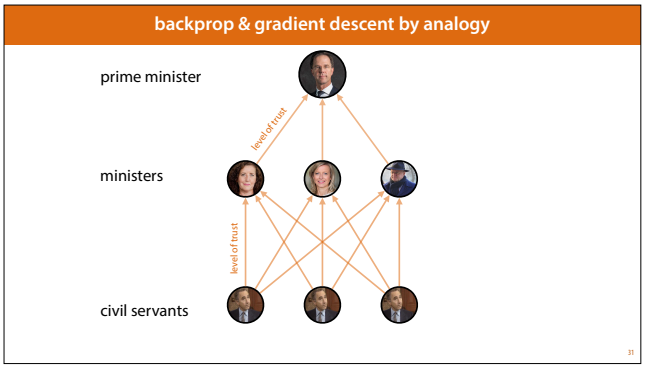
$$\frac{\partial l}{\partial v_2} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial v_2}$$
$$= 2(y - t) \cdot h_2$$

$$v_2 \leftarrow v_2 - \eta \cdot 2(y - t) \cdot h_2$$

30

Here's what the local gradients look like for the weight v_2.

The line on the bottom shows how we update v_2 when we apply a single step of stochastic gradient descent for x (x may not appear in the gradient, but the values y and h2 were computed using x).



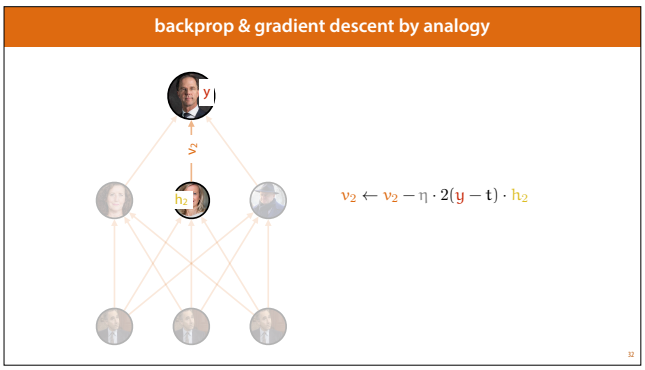
To see what this update rule means, we can use an analogy. Think of the neural network as a hierarchical structure like a government trying to make a decision. The output node is the prime minister: he provides the final decision (for instance what the tax on cigarettes should be).

To make this decision, he listens to his ministers. His ministers don't tell him what to do, they just shout. The louder they shout, the higher they want him to make the output.

If he trusts a particular minister, he will weigh their advice positively, and follow their advice. If he distrusts the minister, he will do the opposite of what the minister says. The ministers each listen to a bank of civil servants and weigh their opinions in the same way the prime minister weight theirs. All ministers listen to the same civil servants, but they have their own level of trust for each.

(We haven't drawn the bias, but you can think of the bias as the prime minister's own opinion; how strong the opinions of the ministers need to be to change his mind).

image sources:
https://www.government.nl/government/members-of-cabinet/mark-rutte
https://www.government.nl/government/members-of-cabinet/ingrid-van-engelshoven
https://www.rijksoverheid.nl/regering/bewindspersonen/kajsa-ollongren
Door Photo: Yordan Simeonov (EU2018BG) - Dit bestand is afgeleid van: Informal JHA meeting (Justice) Arrivals (26031834658).jpg, CC BY-SA 2.0, **https://commons.wikimedia.org/w/index.php?curid=70324317**



So let's say the network has produced an output. The prime minister has set a tax on cigarettes y, and based on the consequences realises that he should actually have set a tax of t. He's now going to adjust his level of trust in each of his subordinates.

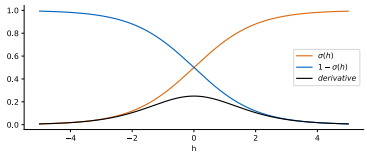
Looking at the update rule for weight v_2, we can see that he takes two things into account: the error (y-t), how wrong he was, and what minister h_2 told him to do.

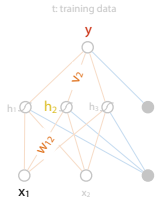
- If the error is positive, he set y too high. If h_2 shouted loudly, he will lower his trust in her.
- If the error is negative, he set y too low. If h_2 shouted loudly, he will increase his trust in her.

If we use a sigmoid activation, the ministers can only provide values between 0 and 1. If we use an activation that allows h2 to be negative, we see that the minister takes the sign into account: if h2 was negative and the error was negative too, the trust in the minister increases (because the PM should've listened to her).

derivative of the sigmoid

$$\frac{\partial \sigma(h)}{\partial h} = \sigma(h)(1 - \sigma(h))$$





$$l = (y - t)^2$$
$$y = v_1 h_1 + v_2 h_2 + v_3 h_3 + b_y$$
$$h_2 = \frac{1}{1 + \exp(-h_2')}$$
$$h_2' = w_{12} x_1 + w_{22} x_2 + b_2$$

$$\frac{\partial l}{\partial w_{12}} = \frac{\partial l}{\partial y} \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial h_2'} \frac{\partial h_2'}{\partial w_{12}}$$

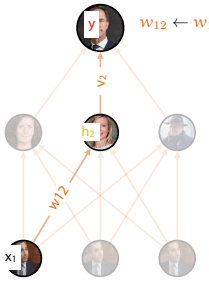
$$= 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

$$w_{12} \leftarrow w_{12} - \eta \cdot 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

So far, this is no different from gradient descent on a linear model. The real power of the backpropagation algorithm shows when we look at how the error propagates back down the network (hence the name) and is used to update the weights. Lets look at the derivative for weight w12

---

backpropagation by analogy



$$w_{12} \leftarrow w_{12} - \eta \cdot 2(y - t) \cdot v_2 \cdot h_2(1 - h_2) \cdot x_1$$

global error

h_2's error

h_2''s contribution, considering the activation function

x_1's contribution

To see how much minister h_2 needs to adjust her trust in x_1, she first looks at the *global error*. To see how much she contributed to that global error, and whether she contributed negatively or positively, she multiplies by v_2, her level of influence over the decision. Then she looks at how much the input from all her subordinates influenced the decision, considering the activation function (that is, if the input was very high, she'll need a bigger adjustment to make a meaningful difference). Finally she multiplies by x_1, to isolate the effect that we trust in x_1 had on her decision.

---

summary

Neural networks are networks of artificial neurons: linear units with nonlinear activations.
These are **very** lightly inspired by actual neurons

The simplest version is the feedforward network.

Trained by stochastic gradient descent.

To avoid working out the entire gradient, we use **backpropagation**: work out local derivatives, and fill in the values from the forward pass.

'90–'95: the start of the neural network winter

Why did interest in Neural Networks die out?

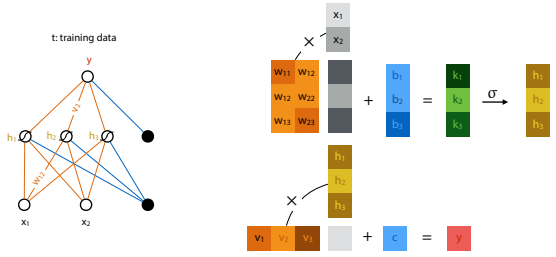
- NNs are non-convex, difficult to train.
  SVMs have convex loss, optimal solution guaranteed
- A single forward backward pass was very expensive.
  Many passes required to train an NN
- They're not that great as classifiers/regression models
  They're not bad, but their power is in their versatility (more next week).
- We didn't have good libraries/abstractions. Too many possibilities.

37

These weren't just reasons not to use neural nets in production. They also slowed down the research on neural nets. SVM researchers were (probably) able to more faster, because once they'd designed a kernel, they could compute the optimal model performance and know, without ambiguity, whether it worked or not. Neural net researchers could design an architecture and spend months tuning the training algorithm without ever knowing whether the architecture would eventually perform.

---

next lecture: it's all just linear algebra



$$f(\mathbf{x}) = \mathbf{V}\,\sigma(\underbrace{\mathbf{W}\mathbf{x}+\mathbf{b}}_{\mathbf{h}})+\mathbf{c}$$

38

One important part of building such a framework is to recognise that all of this can easily be described as matrix multiplication/addition, together with the occasional element-wise non-linear operation. This allows us to write down the operation of a neural network very elegantly.

In order to make proper use of this, we should also work out how to do the backpropagation part in terms of matrix multiplications. That's where we'll pick up next week in the first **deep learning** lecture.

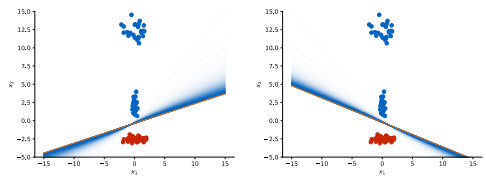
Beyond Linear Models

Part 3: Support Vector Machines

Machine Learning 2020
mlvu.github.io
Vrije Universiteit Amsterdam

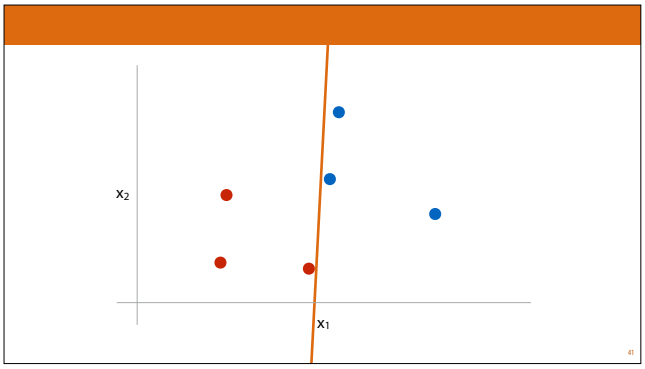
---

logistic



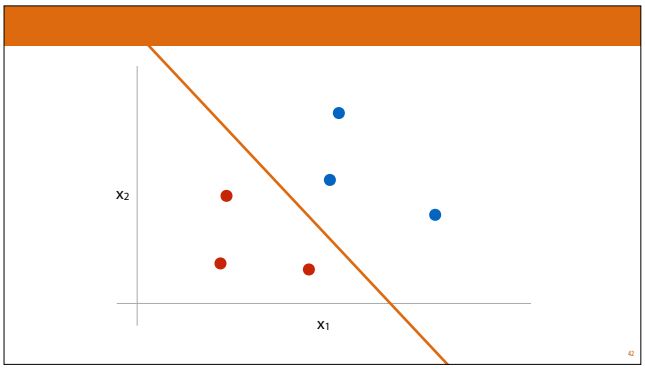
40

In lecture 5, we introduced the logistic regression model, with the logarithmic loss. We saw that it performed very well, but it had one problem: when the data are very well separable, it didn't have any basis to choose between two models like this: both separate the training data very well. Yet, they're very different models.

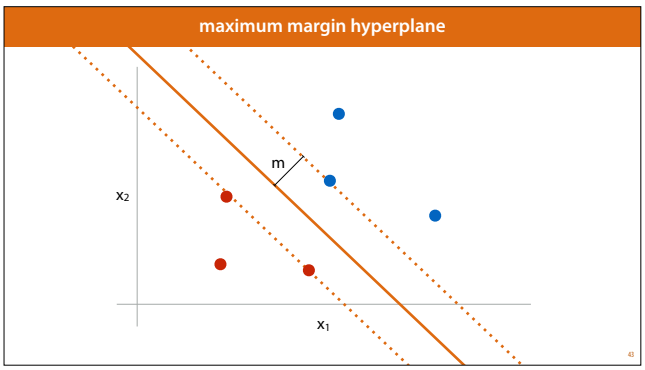


Here is an extreme example of the problem. We have two linearly separable classes and a decision boundary separates the data perfectly. And yet, if I see a new instance that is very similar to the rightmost red point, but with a slightly higher x_1 value, it is suddenly classified as a blue point.

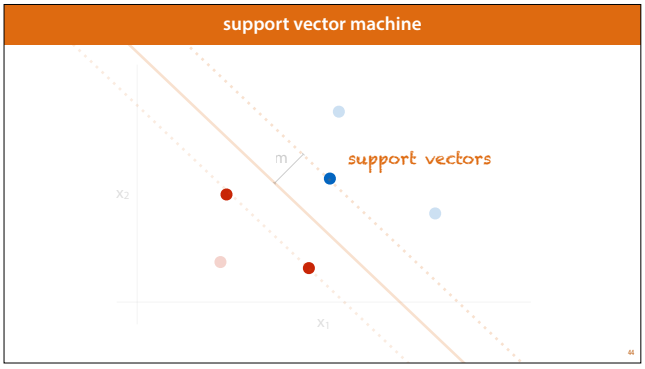
This illustrates the intuition behind our final loss function: if we generate new points *near* our existing points, they should be classified the same as the existing points. One way to accomplish this is to look at the distance from the decision boundary to the nearest red and blue points, and to *maximize* that.



What we are looking for is the hyperplane that has a maximal distance to the nearest positive and nearest negative point. Here's the optimal solution for this data.



We measure the distance m at a right angle to the hyperplane. For the blue class, there is only one point nearest the margin, but for the red class, there are two the same distance away.



The points closest to the decision boundary are called the **support vectors**. This name comes from the fact that the support vectors alone, are enough to describe the model. If I give you the support vectors, you can work out the hyperplane without seeing the rest of the data.

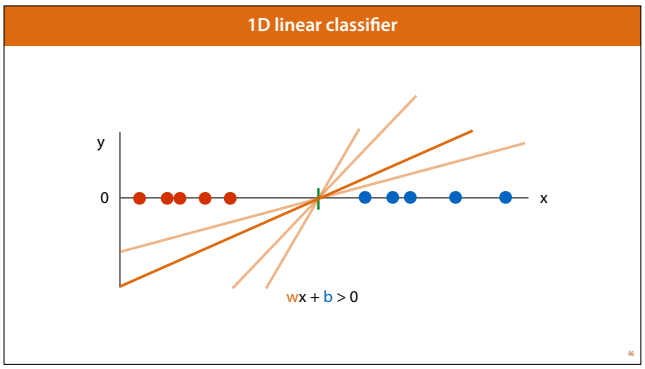
The distance to the support vectors is called the **margin**. We'll assume that the decision boundary is chosen so that the margin is the same on both sides.

Given a dataset, how do we work out which hyperplane maximizes the margin?

So, given a dataset, how do we work out which hyperplane maximizes the margin?

This is a tricky problem, because the support vectors aren't *fixed*. If we move the hyperplane around to maximize the distance to one set of support vectors, we may move to close to other points, making them the support vectors.

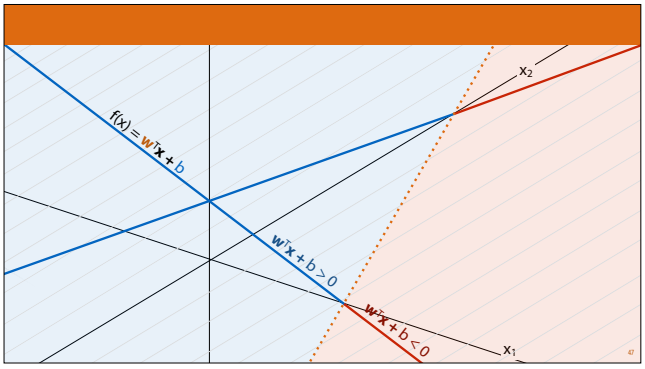
Surprisingly, there is a way to phrase the maximum margin hyperplane objective as a relatively simple optimization problem.



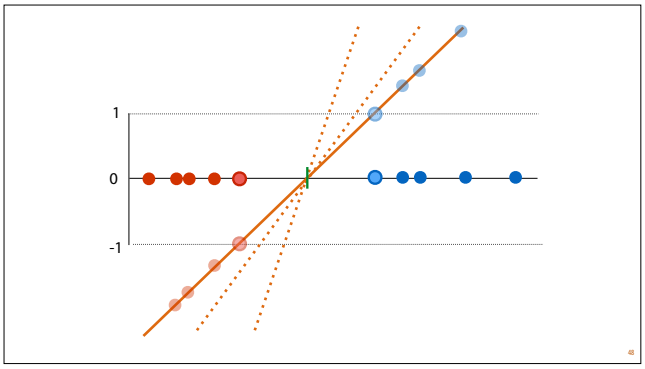
To work this out, let's first review how we use a hyperplane to define a linear decision boundary. Here is the 1D case. We have a single feature and we first define a linear function form the feature space to a scalar y.

If the function is positive we asign the positive class, if it is negative, we assign the negative class. Where this function is equal to 0, where is "intersects" the feature space, is the decision boundary (which in this case is just a point).

Note that by defining the decision boundary this way, we have given ourselves an extra degree of freedom: the same decision boundary can be defined by infinitely many hyperplanes. We'll use this extra degree to help us define a single hyperplane to optimize.

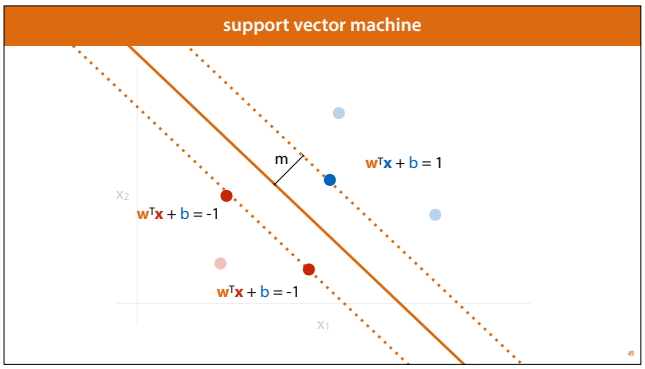


Here's the picture for a two dimensional feature space. The decision boundary is the dotted line where the hyperplane intersects the (x_1, x_2) plane. If we rotate the hyperplane about that dotted line, we get another hyperplane defining the same decision boundary.



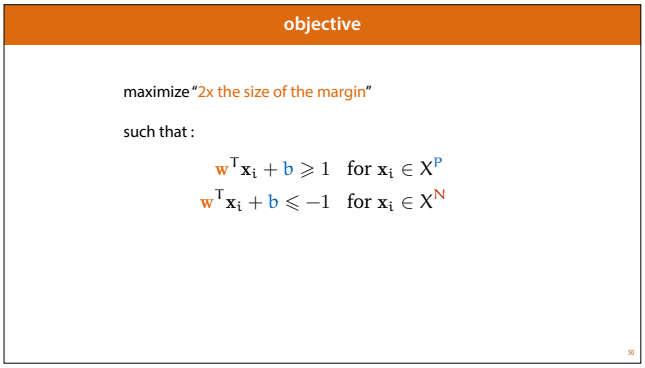
The hyperplane h we will choose is the one that produces 1 for the positive support vectors and -1 for the negative support vectors. Or rather, we will *define* the support vectors as those points for which the line produces 1 and -1.

For all other negative points, h should produce values below -1 and for all other positive points, h should produce values *above* 1.



This is the picture we want to end up with in 2 dimensions. The linear function evaluates to -1 the negative suport vectors, and to a lower value for all other negative points. It evaluates to 1 for the positive support vectors.

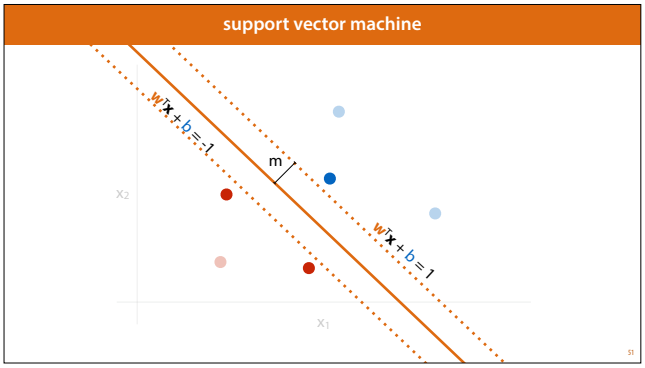
The trick we use to achieve this is to optimize *with a constraint*. We first define the margin as the distance from the decision boundary, where h evalutez to zero, to the line where h evaluates to 1, and on the other side to the line where h evaluates to -1.



Here is our objective. The quantity that we want to maximize is 2 times the margin: the width of the band separating the negative from the positive support vectors.

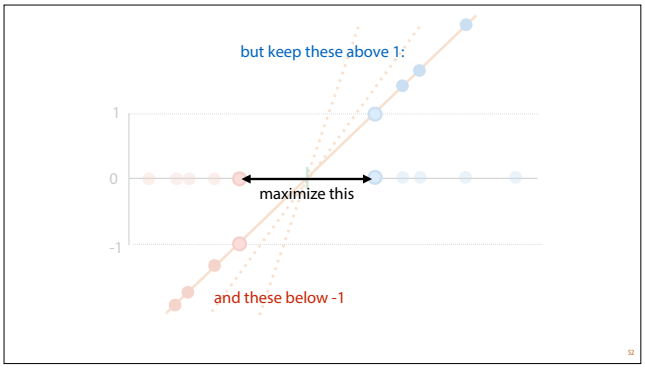
The constraint defines the support vectors: all positive points should evaluate to 1 or higher. All negative points should evaluate to -1 or lower. Note that if we have N instances in our data, this gives us a problem with N constraints.

Note that this automatically ensures that the support vectors end up at -1 and 1. Why?

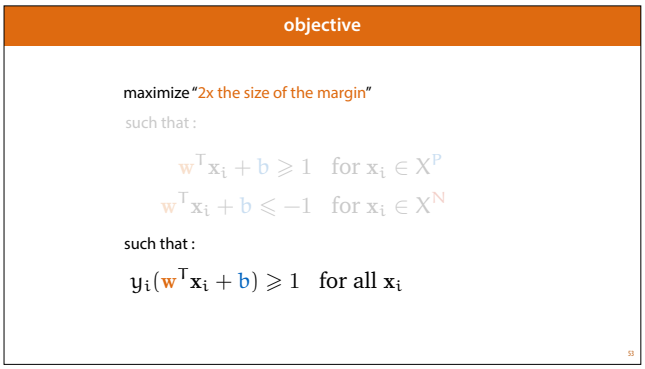


Here is a picture of a case where all nagative points are strictly less than -1, and all positive points are strictly large than 1. the constraints are satisfied, bu there are no points on the edges of the margin.

In this case, we can easily make the margin bigger, pushing it out to the support vectors. Thus, any hyperplane with a maximal margin, that satisfies the constraints. must have points on the edges of its margin. These points are the support vectors.



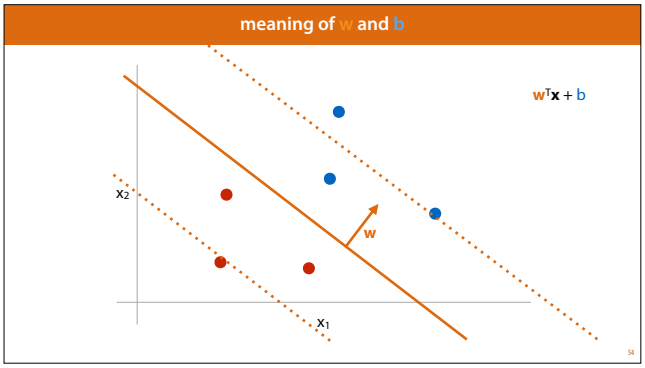
Here is the picture for a single feature. We want to maximize the distance between the point where the hyperplane hits -1 and where it hits 1, while keeping the negatives below -1 and the positives above 1.



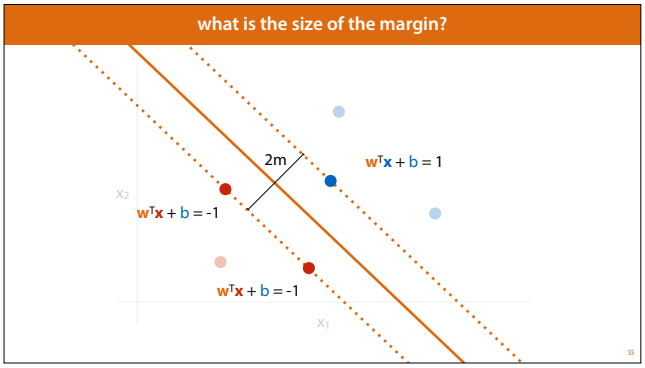
The first thing we'll do, it to simplify the two constraints fro different points into one.

We introduce a a label y_i for each point x_i which is -1 for negative points and +1 for positive points. Multiplying the left-hand side of the constraint by y_i keeps it the same for positive points and takes the negative for negative points. This means that in both case, the left hand side should now be larger than or equal to one. This is the same label we introduced to define the least squares loss.

We now have a problem with the same constraint of every instance in the data.

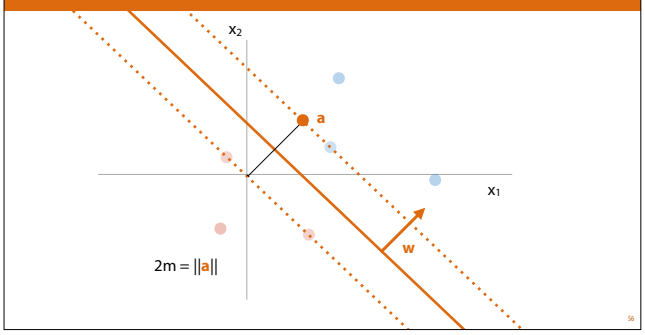


First, remember that in the equation $w^T x + b$, w is the vector pointing orthogonally to the decision boundary.



This is the value we're insterested in. Twice the margin.

what is the size of the margin?



To make the math easier, let's move the axes around so that the lower dotted line (belonging to the negative support vectors) crosses the origin. Doing this doesn't change the size of the margin.

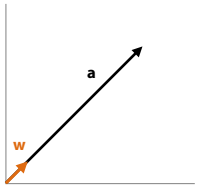
We can now imagine a vector from the origin to the upper dotted line, which we'll call **a**. The length of this vector is exactly the quantity we're interested in.

Remember also that the vector **w** points in the same direction as **a**, because both are perpendicular to the decision boundary.

---

what is the size of the margin?

$$\mathbf{w}^T\mathbf{0} + b = -1$$
$$\mathbf{w}^T\mathbf{a} + b = 1$$
$$\mathbf{w}^T\mathbf{a} = 2$$
$$||\mathbf{w}||\,||\mathbf{a}|| = 2$$
$$||\mathbf{a}|| = \frac{2}{||\mathbf{w}||}$$



$$\mathbf{w}^T\mathbf{x} = ||\mathbf{w}||\,||\mathbf{x}||\cos\alpha$$

Because of the way we've moved the hyperplane, we know that the origin (0) hits the negative margin, so evaluates to -1. We also know that a hits the positive margin, so evaluates to +1.

Subtracting the first from the second, we find that the dot product of **a** and **w** must be equal to two.

Since **a** and **w** point in the same direction (cos α = 1), their dot product is just the product of their magnitudes (see the geometric definition of the dot product on the left).

Re-arranging, we find that the length of **a** is 2 over that of **w**.

---

objective

maximize :

$$\frac{2}{||\mathbf{w}||}$$

such that :

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geqslant 1 \quad \text{for all } \mathbf{x}_i$$

So, the thing we actually want to maximise is 2/||**w**||. This gives us a precise optimization objective.

Note that almost all the complexity of the loss is in the constraints. Without them we could just let all elements of **w** go to zero However, the constraints require the output of our model to be larger than 1 for all positive points and smaller than -1 for all negative points. This will automatically push the margin up to the support vectors, but no further.

objective: **hard margin SVM**

minimize : $\frac{1}{2}\|w\|$

such that :

$$y_i(w^T x_i + b) \geqslant 1 \quad \text{for all } x_i$$

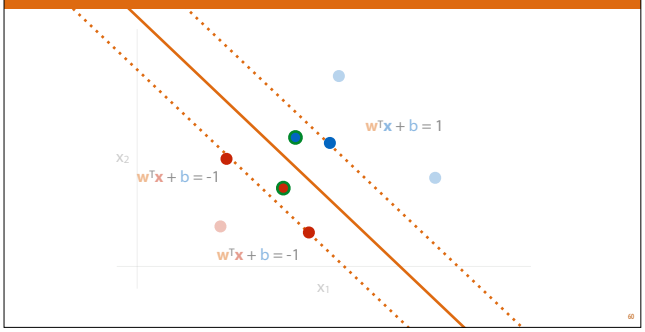
Since we tend to work with loss function, we take the inverse, and minimize that. The resulting classifier is called a "**hard margin**" **support vector machine** (SVM), since no points are allowed to violate the constraint and end up inside the margin.

The hard margin SVM is nice, but it doesn't work well when:

- We have data that is not linearly separable
- We could have a very nice decision boundary if we just ignored a few misclassified points. For instance, when there is a little noise, or a few outliers.

---

soft margin



To deal with such situations, we can allow a **soft margin**. Here we allow a few points to be on the wrong side of the margin, if it helps us achieve a better fit on the rest of the points. That is, we can trade off a few violations of the constraints against a bigger margin.

---

soft margin SVM

minimize : $\frac{1}{2}\|w\| + C\sum_i p_i$

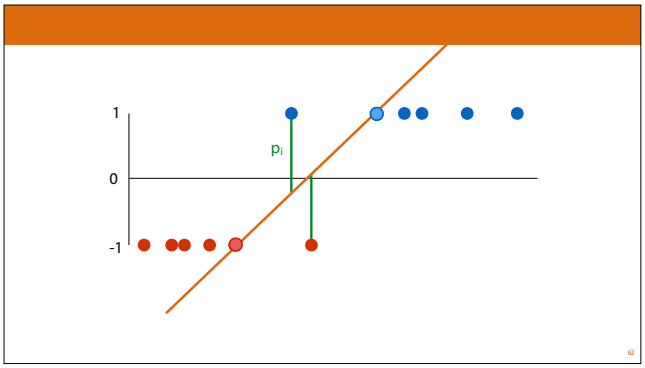
such that :

$$y_i(w^T x_i + b) \geqslant 1 - p_i \quad \text{for all } x_i$$
$$p_i \geqslant 0$$

To achieve this, we introduce a **slack parameter** p_i for each point x_i which indicates the extent to which the constraint on x_i is *relaxed*. Our learning algorithm can set p_i to whatever it likes. If it sets p_i to zero the constraint is the same as it was for the hard margin classifier. If it sets p_i higher than zero, the constraint is relaxed and the point x_i can fall inside the margin. The price we pay is that p_i is added to our minimization objective.

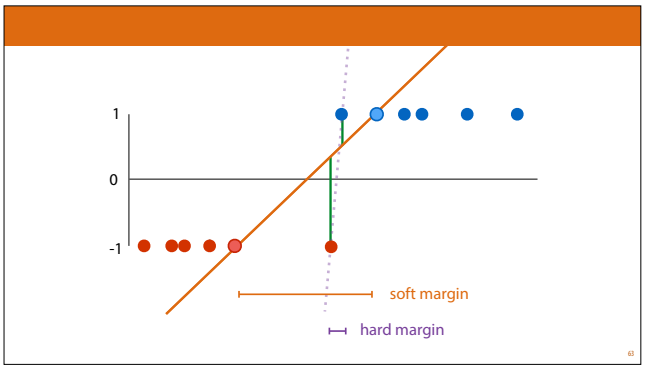
Our search algorithm, which we will detail later, does the rest. It automatically makes the tradeoff between how much we want to violate the original constaints and how big we want the margin to be.

C is a hyperparameter, indicating how to balance the tradeoff. Its value is positive, and we usually try values like (0.001, 0.01, 0.1, 1.0, 10). As C goes to infinity, we recover the hard margin SVM, where violating the constraints is "infinitely bad" and this never happens.



Here is what that looks like in 1D. The open points are the support vectors, and for each class, we have one point on the wrong side of the decision boundary, requiring us to pay the residual p^i as a penalty.

So, the objective function has a penalty added to it, but without this penalty, we would not have been able to satisfy the objective at all, since the two points are not separable



Even if the points *are* linearly separable, it can be good to allow a little slack.

Here, the two points that would be the support vectors of the hard margin objective leave a very narrow margin. By allowing a little slack, we can get a much wider margin that provides a decision boundary that is more likely to generalise to unseen data.

How do we search for a good solution to the soft margin SVM objective?

So, now that we have made our objective precise, how do we find a good solution? We haven't discussed constrained optimization much yet.

a fork in the road

option one: express everything in terms of **w**, get rid of the constraints. this video

- Allows gradient descent to be used.
- Good for use with neural networks/deep learning.

option two: express everything in terms of the support vectors, get rid of **w**. next video

- Doesn't allow error to propagate back, but …
- allows the **kernel trick** to be applied.

Here we have to options. One allows us to use the old familiar gradient descent, without having to worry about constraints.

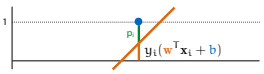
The other requires us to delve into constrained optimization, which we will do in the next video. The payoff for that is that it opens the door to the **kernel trick**.

In the rest of this video, we will work out option one.

option one: get rid of the constraints

if $y_i(w^T x_i + b) < 1$:

$p_i = 1 - y_i(w^T x_i + b)$



otherwise:

$p_i = 0$

$1 - y_i(w^T x_i + b)$ is negative

$$p_i = \max\left(0, 1 - y_i(w^T x_i + b)\right)$$

To get rid of the constraints, let's look at what we know about the value of p_i.

If the constraint for $\mathbf{x}_i$ is violated, we can see that p_i makes up the difference between what $y^i(\mathbf{w}^T\mathbf{x}^i + b)$ should be (1) and what it is.

If the constraint is not violated, p_i becomes zero, and the value we computed above becomes negative.

We can summarise these two conclusions in a single definition for p_i. We can then simply fill this definition into the minimization objective, in place of p_i.

---

option one

minimize :

$$\frac{1}{2}\|w\| + C\sum_i \max\left(0, 1 - y_i(w^T x_i + b)\right)$$

regulariser — error

This gives us an **unconstrained loss function** we can directly apply to any model. For instance when training a neural network to classify, this makes a solid alternative to cross-entropy loss. This is sometimes called the **L1-SVM** (loss). There is also the **L2-SVM** loss where a square is applied to the p_i function, to increase the weight of outliers.

We can think of the first term as a *regulariser*. It doesn't enforce anything about how well the plane should fit the data. It just ensures that the parameters of the plane don't grow too big.

The highlighted part of the second term functions as a kind of error (just as we used in least squares classification, but without the square). It computes how far the model output of $y_i(\mathbf{w}^T\mathbf{x}_i + b)$ is away from the desired value (1).

However, unlike the least squares classifier, we *only* compute this error for points that are sufficiently close to the decision boundary, for any points far from the boundary, we do not compute any error at all. This is achieved by cutting off any negative values. We could easily put all points far away from the decision boundary, and make all their error terms zero, but this (usually) requires a $\mathbf{w}$ with a high norm, so then the regulariser increases the loss massively.

---

classification losses

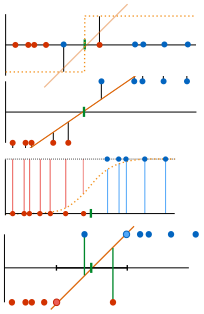
~~Least squares loss (today)~~

~~Log loss / logistic regression (week 3, Probability 1)~~

~~SVM loss (week 3, Linear Models 2)~~

And with that, we have discussed our final classification loss. Let's review.

loss functions



error
aka zero-one loss, nr. of misclassified instances
doesn't work with gradient descent

least squares
assign -1, 1 to points treat as regression
doesn't really work well, illustrative

log-loss
aka cross entropy, use a sigmoid to turn the linear output into probabilities
works well for non-separable data, has multiple solutions, generalizes to multiclass

soft margin SVM
aka Hinge loss, maximum margin loss
works well in high-dim data, separable data

Here are all our loss functions. The error, also known as zero one loss, is imply the number or proportion of misclasssified examples. It's usually what we're interested in, but it doesn't give us a loss surface that is suitable for searching.

Beyond Linear Models
Part 4: Lagrange multipliers

Machine Learning 2020
mlvu.github.io
Vrije Universiteit Amsterdam

In the last video, noted that there are two ways to deal with the maximum margin hyperplane objective.

---

a fork in the road

option one: express everything in terms of **w**, get rid of the constraints. — last video

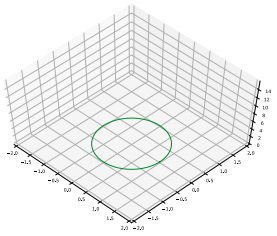
- Allows gradient descent to be used.
- Good for use with neural networks/deep learning.

option two: express everything in terms of the support vectors, get rid of **w**. — this video & the next

- Doesn't allow error to propagate back, but …
- allows the **kernel trick** to be applied.

In the last video, we used some clever tricks to remove the constraints. This allows for hinge loss to be solved with plain gradient descent, and for it to be used as the last layer in a neural network, with the loss backpropagating though the network.

In this video we'll take another tack: we'll keep the constraints, and rewrite the loss function so that **w** and b disappear. This requires us to use constrained optimization, so we'll need to look beyond plain gradient descent, and it doesn't allow us to use the SVM inside a neural network anymore, but if we pay this price, we are rewarded with the ability to use **the kernel trick**, which has some very powerful consequences.

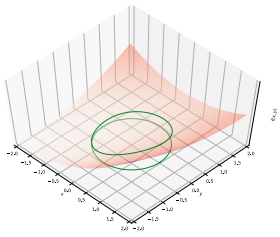


minimize $f(x, y) = x^2 + \frac{1}{2}y^2 - xy + 2x + 2$

such that $x^2 + y^2 = 1$

So we start with optimization under constraints.

First let's make it a little more intuitive what optimization under constraints looks like. Here, we have a simple constrained optimization problem. In this case, the constraint specifies that the solution must lie on the unit circle (that is, x and y must make a unit vector). Within that set of points, we want to find the lowest point on some parabola in x and y.



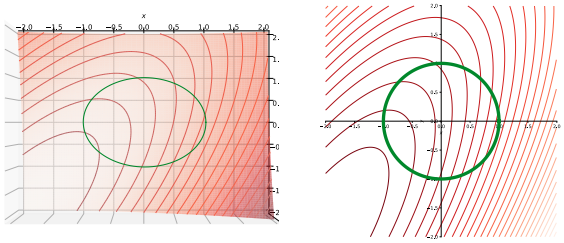
minimize $f(x, y) = x^2 + \frac{1}{2}y^2 - xy + 2x + 2$

such that $x^2 + y^2 = 1$

Projecting the constraints onto our function gives us the set of possible points over which we want to minimize: we want to find the lowest point on this circle.

---

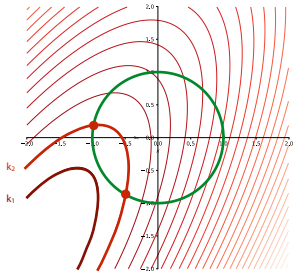
contour maps



One way to help us visualize this problem is to draw iso-lines, also known as contours for the function f. These are the curves that indicate where the function is equal to some constant k. By increasing k, we get different contour lines for the function.

If we look down onto the xy plane, the z axis disappears, and we get a 2D plot of our function, where the contour lines give us an idea of the height of the function. This principle is often used in maps to indicate elevation

---

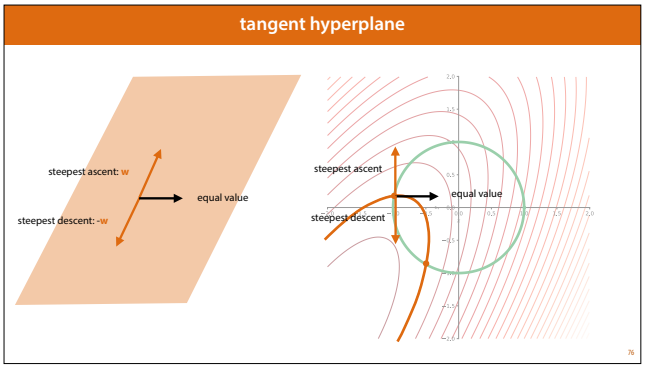


Note that the function f gets lower towards the bottom left corner.

Each contour line indicates a constant output value for our function: the value that we want to minimize. We can tell by this plot what we can achieve.

The output value k_1 is very low (the lowest of the contour lines in this plot), so it would make a very good solution, but it never meets the circle representing our constraints. That means that we can't get the output as low as k_1 *and* satisfy the constraints.

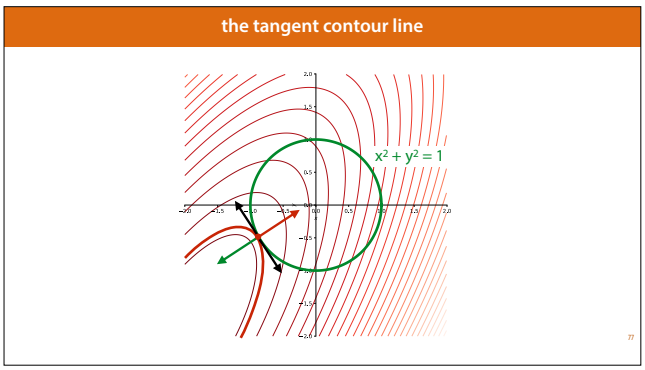
The next lowest output value, k_2, does give us a contour line that crosses the circle representing our constraints. Therefore, we can satisfy our constraints and get at least as low as k_2. However, the fact that it *crosses* the circle of our constrains, means that we can also get *lower* than k_2. Why?



We can work this out based on what we know already: if we have a hyperplane defined by $\mathbf{w}^T\mathbf{x} + b$, then we know that $\mathbf{w}$ is the direction of steepest ascent, and $-\mathbf{w}$ is the direction of steepest descent. This tells us that the direction orthogonal to the line of $\mathbf{w}$ is the direction in which the value of the plane doesn't change: the direction of equal value.

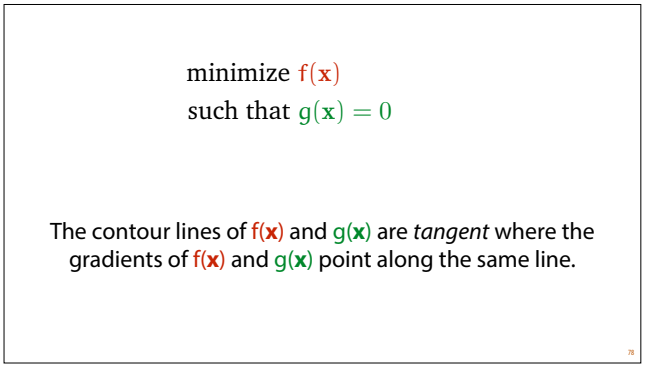
This means that if we take any point on a contour line and work out the tangent hyperplane of f at that point, that is, compute the gradient, the direction orthogonal to the gradient points *along the contour line*.

In this case, since our contour line *crosses* the circle of the constraints, the direction of equal value doesn't point along the circle for the constraints, and we can conclude that the value of f decreases in one direction along the circle and increases in one direction.



By this logic, the only time we can be sure that there is no lower to go along the circle is when the direction of equal value points along the circle. In other words, when the contour line is tangent to the circle: it touches it only at one point without crossing it.

How do we work out where this point is? By recognizing that the circle for our constraints is *also* a contour line. A contour of the function $x^2 + y^2$, for the constant value 1. This means that when the gradient of $x^2 + y^2$ points in the same or opposite direction as that of f, their contour lines are tangent. And when that happens we have a minimum or maximum for our objective.



If we define our problem in general with an objective function $f(\mathbf{x})$ that we want to minimize and a constraint function $g(\mathbf{x})$ that we need to stay equal to zero, then we can state the point we're looking for in terms of their gradients, as shown here.

$$\nabla f(\mathbf{x}) = \alpha \nabla g(\mathbf{x})$$

$$\nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x}) = 0$$
$$\nabla \left(f(\mathbf{x}) - \alpha g(\mathbf{x}) \right) = 0$$

$$\nabla L(\mathbf{x}, \alpha) = 0 \quad \text{with } L(\mathbf{x}, \alpha) = f(\mathbf{x}) - \alpha g(\mathbf{x})$$

We are looking for gradients pointing in the same (or opposite) direction, but not necessarily of the same size. To state this formally, we say that there must be some alpha, such that the gradient of f is equal to the gradient of g *times alpha*.

We rewrite to get something that must be equal to zero. By moving the gradient symbol out in front (the opposite of what we usually do with gradients), we see that what we're looking for is the point where the gradient of some function is equal to zero. This new function we will call L. It is a function is x and the additional parameter alpha.

With that, we've turned our constrained optimization problem into an unconstrained one. The price we pay is one additional parameter to optimize.

---

optimising under constraints

minimize $f(\mathbf{x})$
such that $g(\mathbf{x}) = 0$

$$L(\mathbf{x}, \alpha) = f(\mathbf{x}) - \alpha g(\mathbf{x})$$

solve $\nabla L(\mathbf{x}, \alpha) = 0$ for $\mathbf{x}$ and α

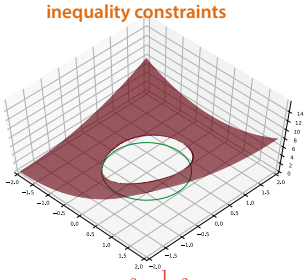
This is the idea of Lagrange multipliers. We rewrite the problem so that he contraints are some function that needs to be equal to zero. Then we create a new function L, which consists of f with alpha times g subtracted. For this new function **x** and alpha are both parameters. Then, we solve for both **x** and alpha

This new function L has an optimum where the original function is minimal within the constraints. The new optimum is a **saddlepoint**: it's a minimum in x and a maximum in alpha. This means can't solve by basic gradient descent, we have to set its gradient equal to zero, and solve analytically.

If we have multiple constraints, we add multiple terms

---

inequality constraints

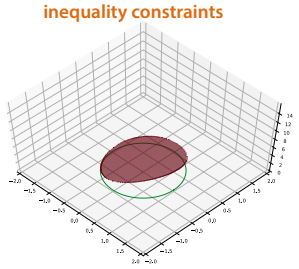


minimize $f(x, y) = x^2 + \frac{1}{2}y^2 - xy + 2x + 2$
such that $x^2 + y^2 \geqslant 1$

If the constraint in our problem is not an equality constraint, but an inequality constraint, we need to keep a few more things in mind.

In a case like this one, the constraint is **inactive**: the constraint doesn't remove the global minimum from the graph, so the solution to the problem is the same as the unconstrained one.

inequality constraints



minimize $f(x, y) = x^2 + \frac{1}{2}y^2 - xy + 2x + 2$

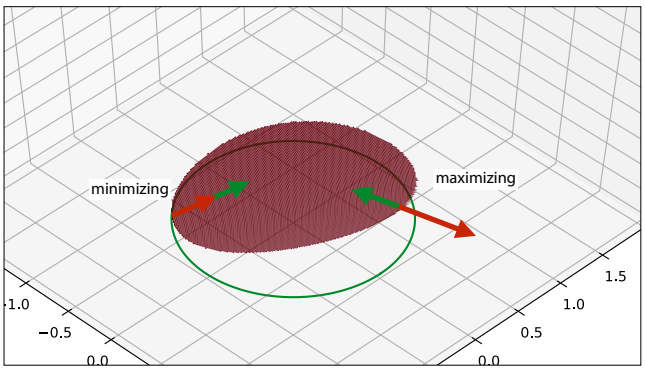
such that ~~$x^2 + y^2 \leqslant 1$~~ $-x^2 - y^2 + 1 \geqslant 0$

If we search only inside the unit circle, the constraint is **active**. It stops us from going where we want to go, and we end up on the boundary, just like we would if the constraint were an equality constraint.

We first set the convention that all constraints are rewritten to be "greater than" inequalities, with zero on the right hand side. This doesn't change the region we're constrained to, but note that the function on left of the inequality sign had a "bowl" shape before, and now has a "hill" shape. In other words, the gradients of this function now point in the opposite direction.

With that, there is one important thing to keep in mind, compared to the Lagrange multipliers

---



If we are *minimizing*, we need to make sure that the gradient points **into** the constrained region, so that the direction of steepest descent points outside. If the direction of steepest descent pointed into the region, we could find a lower point somewhere inside, away from the boundary. Since the graident for the constraint function points inside the region, we need to make sure that the gradients point in the same direction.

If we are *maximizing*, by the same logic, we need to make sure that the gradients point in opposite directions.

Lagrangian dual

$$\nabla f(\mathbf{x}) = \alpha \nabla g(\mathbf{x}) \text{ with } \alpha \geqslant 0$$

$$\nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x}) = 0$$
$$\nabla \left(f(\mathbf{x}) - \alpha g(\mathbf{x}) \right) = 0$$

$$\nabla L(\mathbf{x}, \alpha) = 0 \text{ with } L(\mathbf{x}, \alpha) = f(\mathbf{x}) - \alpha g(\mathbf{x}) \text{ and } \alpha \geqslant 0$$

This makes the derivation a little more complicated: we again set the gradient of the objective function equal to that of the constraint, again with an alpha to account for the differences is size between the two gradients, but this time around, we need to make sure that alpha remains positive, since a negative alpha would cause the gradient of the constraint to point in the wrong direction.

We work out L as before, but we don't end up with an unconstrained problem. We've simply traded one constrained problem for another. This new problem is sometimes called the **Lagrangian dual** of the original problem. A dual is a mathematical term for a different way of phrasing something.

Even though we've not removed the constraint, we've simplified it a lot: it is now a linear function, even a constant one, instead of a nonlinear function. Linear constraints are much easier to handle, for instance using methods like linear programming, or gradient descent with projection. The price we've paid for this easier problem is an extra

KKT multipliers

$$\text{minimize } f(\mathbf{x})$$
$$\text{such that } g(\mathbf{x}) \geqslant 0$$

$$L(\mathbf{x}, \alpha) = f(\mathbf{x}) - \alpha g(\mathbf{x})$$
$$\text{solve } \nabla L(\mathbf{x}, \alpha) = 0 \text{ for } \mathbf{x} \text{ and } \alpha \geqslant 0$$

Here then, is the method of **KKT multipliers**. It's just like the Lagrangian method with two points we need to keep in mind:

- For a "greater than 0" inequality and a minimization problem we subtract the constraint term from the objective function. If we maximize instead of minimize we add the term.
- The resultant problem has one constraint, which is that the KKT multiplier alpha must be positive.

Lagrange multipliers

Solve constrained optimization analytically

Or: *rewrite* constrained optimization problems
Results in a different, equivalent problem that may teach us something, or be easier to solve.

Multiple constraints: repeat the process.
Every constraint adds one term to L, with one additional Lagrange/KKT multiplier.

Plain gradient descent doesn't solve $\nabla L(\mathbf{x}, \alpha) = 0$
Some methods exist. See eg. Platt NIPS 1988.

In the next video, we will return to our constrained optimization objective and apply the KKT method to work out the Lagrangian dual. As we will see, this will allow us to get rid of all parameters except the KKT multipliers

Beyond Linear Models
Part 5: The kernel trick

Machine Learning 2020
mlvu.github.io
Vrije Universiteit Amsterdam

In the last video, noted that there are two ways to deal with the maximum margin hyperplane objective.

option two: kernel SVM

$$\text{minimize}: \frac{1}{2}\|w\| + C\sum_i p_i$$

such that :

$$y^i(w^T x^i + b) \geqslant 1 - p^i \quad \text{for all } x^i$$
$$p^i \geqslant 0$$

Here is the original optimization objective again, before we started rewriting. We will use the method of Lagrange multipliers to rewritethis objective to its *Lagrangian dual*.

KKT multipliers

$$\text{minimize } f(x)$$
$$\text{such that } g(x) \geqslant 0$$

$$L(x, \alpha) = f(x) - \alpha g(x)$$
$$\text{solve } \nabla L(x, \alpha) = 0 \text{ for } x \text{ and } \alpha \geqslant 0$$

In the last video we learned how to solve a problem with an inequality constraint. That's just what we have here, except that we have more than one constraint.

The solution is simple, we add one term to L for every constraint in our problem.

---

for inequalities

$$\text{minimize } f(x)$$
$$\text{such that } g_i(x) \geqslant 0 \quad \text{for } i \in [1, n]$$

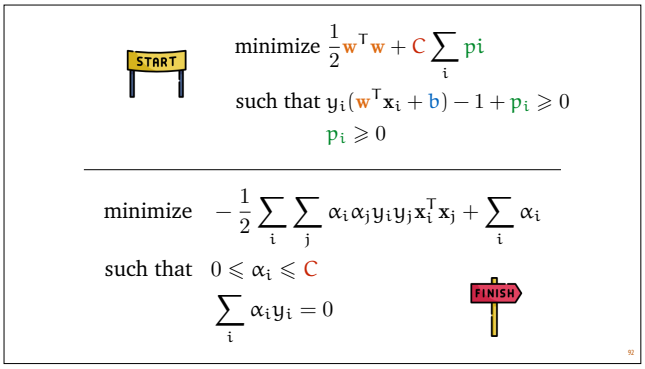
$$L(x, \alpha_1, \ldots, \alpha_n) = f(x) - \sum_i \alpha_i g_i(x)$$

$$\text{solve } \nabla L = 0$$
$$\text{such that } \alpha_i \geqslant 0 \quad \text{for } i \in [1, n]$$

Here's what that looks like. Each term gets its own KKT multiplier α_i, and for all of these, we get a new constraint that it should be positive.

$$\text{minimize } \frac{1}{2}\|w\| + C\sum_i p_i \qquad \|w\| = \sqrt{w^T w}$$
$$\text{such that } y_i(w^T x_i + b) \geqslant 1 - p_i$$
$$p_i \geqslant 0$$

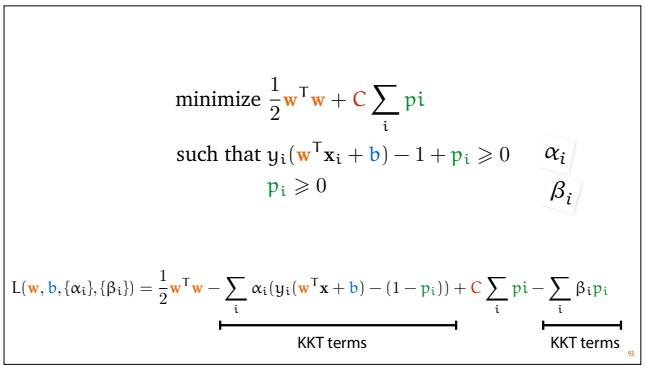
$$\text{minimize } \frac{1}{2} w^T w + C\sum_i p_i$$
$$\text{such that } y_i(w^T x_i + b) - 1 + p_i \geqslant 0$$
$$p_i \geqslant 0$$



The derivation of our new objective is a long and complicated one, so let's first set up what we plan to achieve. The objective above the line is the one we defined in the last video.

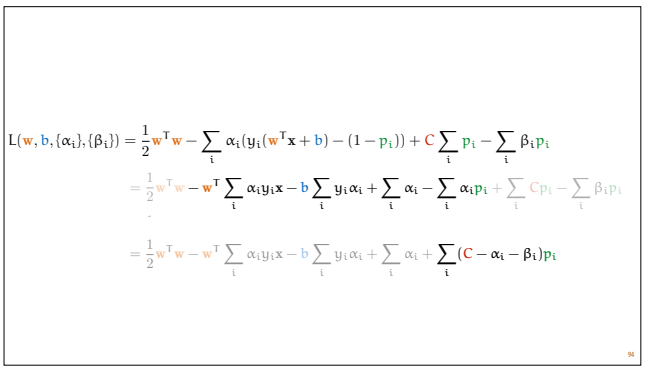
We want to show, by the method of KKT multipliers that this objective is equivalent to the objective below the line.

image source: https://www.flaticon.com/free-icons/road



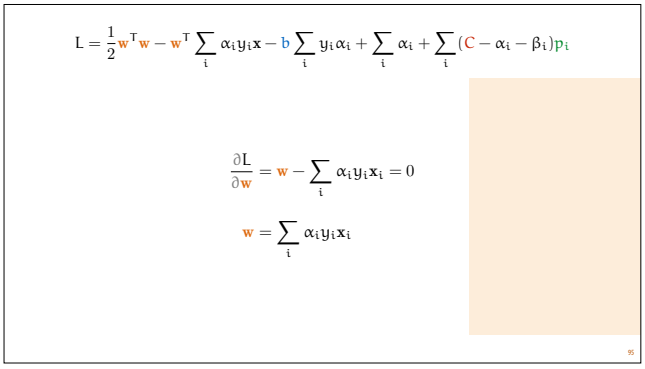
First, we define our L function: the optimization objective with terms added for all inequality constraints..

Note that we've changed the optimization objective from the size of the margin to its square ,which is equal to the dot product of **w** with itself. This doesn't change the location of the optimum, and it makes the solution easier to work out.



We start by writing out our optimization objective.

We muliply out the alphas, and then do the opposite for the p's, taking them outside the brackets.



For this function, we take the derivative with respect to the parameters, and set them equal to zero. We haven't discussed taking derivatives with respect to vectors, but here we'll just use two rules that are analogous to the way we multiply scalars.

- The derivative of the dot product of **w** with itself, wrt to **w** is 2 times **w**. This is analogous to the derivative of the square for scalars.
- The derivative of **w** times some constant vector (wrt to w) is just that constant. This is similar to the constant multiplier rule for scalars.

This gives us an expression for **w** at the optimum, in terms of alpha, y and **x**,

$$L = \frac{1}{2} w^T w - w^T \sum_i \alpha_i y_i x - b \sum_i y_i \alpha_i + \sum_i \alpha_i + \sum_i (C - \alpha_i - \beta_i) p_i$$

$$w = \sum_i \alpha_i y_i x_i$$

$$\frac{\partial L}{\partial b} = -\sum_i y_i \alpha_i = 0$$

$$\sum_i y_i \alpha_i = 0$$

If we take the derivative with respect to b, we find a simple constraint: that at the optimum, the sum of all alpha values (multiplied by their corresponding y's) should be zero.

$$L = \frac{1}{2} w^T w - w^T \sum_i \alpha_i y_i x - b \sum_i y_i \alpha_i + \sum_i \alpha_i + \sum_i (C - \alpha_i - \beta_i) p_i$$

$$w = \sum_i \alpha_i y_i x_i$$

$$\sum_i y_i \alpha_i = 0$$

$$\frac{\partial L}{\partial p_i} = (C - \alpha_i - \beta_i) = 0$$

$$0 \leqslant \alpha_i \leqslant C$$

Finally, we take the derivative for pi and set that equal to zero.

The result essentially tells us that the alpha plus the beta must equal C. If we assume that alpha is between 0 and C, then we can just take beta to be the remainder.

$$L = \frac{1}{2} w^T w - w^T \sum_i \alpha_i y_i x - b \sum_i y_i \alpha_i + \sum_i \alpha_i + \sum_i (C - \alpha_i - \beta_i) p_i$$

$$w = \sum_i \alpha_i y_i x_i$$

$$\sum_i y_i \alpha_i = 0$$

$$0 \leqslant \alpha_i \leqslant C$$

$$(C - \alpha_i - \beta_i) = 0$$

$$L = \frac{1}{2} w^T w - w^T \sum_i \alpha_i y_i x + \sum_i \alpha_i \qquad 0 \leqslant \alpha_i \leqslant C \qquad \sum_i y_i \alpha_i = 0$$

This (in the orange box) is what we have figured out so far about our function at the optimum.

If we fill in the three equalities, our function simplifies a lot. This function describes the optimum, subject to the constraints on the right. These are constraints of variables in our final form, so we need to remember these.

We can now fill in w to get rid of **w**.

$$L = \frac{1}{2} w^T w - w^T \sum_i \alpha_i y_i x_i + \sum_i \alpha_i \qquad w = \sum_i \alpha_i y_i x_i$$

$$= \frac{1}{2} w^T w - w^T w + \sum_i \alpha_i$$

$$= -\frac{1}{2} w^T w + \sum_i \alpha_i$$

$$= -\frac{1}{2} \left[\sum_i \alpha_i y_i x_i \right]^T \left[\sum_i \alpha_i y_i x_i \right] + \sum_i \alpha_i$$

To simplify our function further, we first replace one of the sums in the first line with w.

Then, we replace the final two occurrences of w, with the sum. This gives us a function purely in terms of alpha, with no reference to w or b. All we need to do is simplify it a little bit.

Note that the square brackets here are just brackets, they have no special meaning.

$$L = -\frac{1}{2}\left[\sum_i \alpha_i y_i \mathbf{x}_i\right]^T \left[\sum_i \alpha_i y_i \mathbf{x}_i\right] + \sum_i \alpha_i \qquad (a + b + c)(x + y)$$

$$= -\frac{1}{2}\left[\sum_i \alpha_i y_i \mathbf{x}_i^T \left[\sum_j \alpha_j y_j \mathbf{x}_j\right]\right] + \sum_i \alpha_i \qquad = (a(x + y) + b(x + y) + c(x + y))$$

$$= -\frac{1}{2}\left[\sum_i \left[\sum_j \alpha_i y_i \mathbf{x}_i^T \alpha_j y_j \mathbf{x}_j\right]\right] + \sum_i \alpha_i \qquad = ((ax + ay) + (bx + by) + (cx + cy))$$

$$= -\frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + \sum_i \alpha_i \qquad = ax + ay + bx + by + cx + cy$$

To simplify, we distirbute all dot products over the sums. Note that the dot product distributed over sums the same way as scalar multiplication: $(a + b + c)^T d$ -> $(a^T d + b^T d + c^T d)$.

It looks a little intimidating with the capital sigma notation, but it's the same thing as you see on the right, except with dot products instead of of scalar mulitplication.

$$\text{minimize } \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_i p_i$$

$$\text{such that } y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 + p_i \geqslant 0$$

$$p_i \geqslant 0$$

$$\text{minimize } -\frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + \sum_i \alpha_i$$

$$\text{such that } 0 \leqslant \alpha_i \leqslant C$$

$$\sum_i \alpha_i y_i = 0$$

With that, we have achieved our objective. What has this bought us? We still need to optimize under constraints, so we still need a fancy optimization algorithm to solve this.

We wont go into the details, but a popular choice for SVMs is the relatively simple sequential minimal optimization (SMO) algorithm, a form of linear programming specifically developed for SVMs.

https://en.wikipedia.org/wiki/Sequential_minimal_optimization

image source: https://www.flaticon.com/free-icons/

$$\text{minimize } -\frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j + \sum_i \alpha_i$$

$$\text{such that } 0 \leqslant \alpha_i \leqslant C$$

$$\sum_i \alpha_i y_i = 0$$

The main result here is twofold:

first, notie that the hyperplane parameters have *disappeared* from the minimization objective. The only parameters that remain are one alpha per instance in our data, and the hyperparameter C. The alphas function as an encoding of the support vectors: any instance for which the corresponding alpha is not zero is a support vector.

The second thing to notice is that the algorithm only operates on the **dot products** of pairs of instances. In other words, if you didn't have access to the data, but I *did* give you the full matrix of all dot products of all pairs of instances, you would still be able to find the optimal support vectors. This allows us to use a very special trick.

the kernel trick

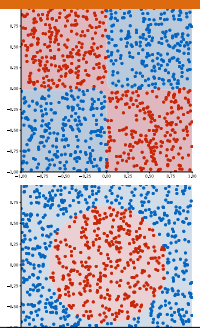
If you have an algorithm which operates only on the dot products of instances, you can substitute the dot product for a kernel function.

What if I didn't give you the actual dot products, but instead gave you a different matrix of values, that *behaved* like a matrix of dot products.

---

making linear models more powerful

d	p	d * p
0.75	0.98	0.74
-0.66	-0.32	0.21
-0.45	0.84	-0.38
0.93	0.78	0.72
-0.42	0.24	-0.10
-0.02	0.43	-0.01
-0.74	0.58	-0.43
-0.41	-0.41	0.17
0.59	0.72	0.42



Remember, by adding features that are derived from the original features, we can make linear models more powerful. If the number of features we add grows very quickly (like if we add all 5-way cross products), this can becomes a little expensive (both memory and time wise)

---

extending the feature space

x_1	x_2	x_1^2	x_1x_2	x_2^2	
3	105	9	315	11025	male
1	110	1	110	12100	male
7	119	49	833	14161	male
8	120	64	960	14400	male
9	120	81	1080	14400	male
12	119	144	1428	14161	female
8	122	64	976	14884	female
8	125	64	1000	15625	female
9	125	81	1125	15625	female
9	132	81	1188	17424	male
14	128	196	1792	16384	female

Let's look at an example. The simplest way we saw to extend the feature space was to add **all cross-products**. This turns a 2D dataset into a 5D dataset.

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$k(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2$$

Here are two 2D feature vectors. What if, instead of computing their dot product, we computed the square of their dot product. It turns out that this is equal to the dot product of two other 3D vectors **a'** and **b'**.

$$(\mathbf{a}^T\mathbf{b})^2 = (a_1b_1 + a_2b_2)^2$$

$$= a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2$$

$$= a_1^2 \cdot b_1^2 + \sqrt{2}a_1a_2 \cdot \sqrt{2}b_1b_2 + a_2^2 \cdot b_2^2$$

$$= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix}$$

The square of the dot product in the 2D feature space, is equivalent to the regular dot product in a 3D feature space.

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$k(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2$$

$$\mathbf{a'}^T\mathbf{b'} = (\mathbf{a}^T\mathbf{b})^2 \qquad \mathbf{a'} = \begin{pmatrix} a_1^2 \\ a_2^2 \\ \sqrt{2}a_1a_2 \end{pmatrix}, \mathbf{b'} = \begin{pmatrix} b_1^2 \\ b_2^2 \\ \sqrt{2}b_1b_2 \end{pmatrix}$$

This function k computes the dot product in a feature space of *expanded* features. We could do this already, but before we had to actually *compute* the new features. Now, all we have to do is compute the dot product in the original feature space and square it.

$$\text{minimize} \quad -\frac{1}{2}\sum_i \sum_j \alpha_i\alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) + \sum_i \alpha_i$$

$$\text{such that} \quad 0 \leqslant \alpha_i \leqslant C$$

$$\sum_i \alpha_i y_i = 0$$

Since the solution to the SVM is expressed purely in terms of the dot product, we can replace the dot product this kernel function. We are now fitting a line in a higher-dimensional space, without computing any extra features explicitly.

Note that this only works because we rewrote the optimization objective to get rid of **w** and b. Since **w** and b have the same dimensionality as the features, keeping them in means using explicit features.

Saving the trouble of computing a few extra features may not sound like a big saving, but by choosing our kernel function cleverly we can push things a lot further.

---

a kernel function

$$k(\mathbf{x}_i, \mathbf{x}_j)$$

A function that computes the dot product of x_i and x_j in a *different* feature space, without explicitly computing those features.

For some expansions to a higher feature space, we can compute the dot product between two vectors, **without explicitly expanding the features**. This is called a **kernel function**.

There are many functions that compute the dot product of two vectors in a highly expanded feature space, but don't actually require you to expand the features.

polynomial kernel

$$k(\mathbf{a}, \mathbf{b}) = \left(\mathbf{a}^T\mathbf{b} + 1\right)^d$$

feature space for d=2: all squares, all cross products, all single features

feature space for d=3: all cubes and squares, all two-way and three-way cross products, all single features.

This is already a **big** feature space.

111

Taking just the square of the dot product, we lose the original features. If we take the square of the dot product plus one, we retain them, and get all cross products. If we increase the exponent to d we get all d-way cross products. Note the main benefit: the d-th power of the dot product + 1 requires almost no more computation than the dot product itself (whatever d). Yet we have now computed the dot product in a feature space with a huge number of extra features.

d is a hyperparameter: increasing it does not make the algorithm much more expensive, but it increases you feature space so much that you seriously risk overfitting.

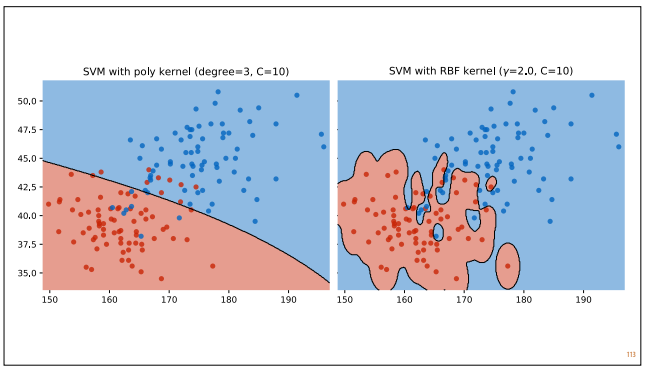
RBF kernel

$$k(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma||\mathbf{a} - \mathbf{b}||\right)$$

feature space: infinite dimensional

112

Gamma is another hyperparameter. The RBF kernel is powerful, but prone to overfitting.



113

Here's a plot for the dataset from the first lecture. As you can tell, the RBF kernel massively overfits for these hyperparameters, but it gives us a very nonlinear fit.

---

kernels in data space

text, DNA, proteins: string kernels (inspired by edit distance)

graphs: Weisfeiler-Lehman distance

114

One of the most interesting application areas of kernel methods is places where you can turn a distance metric in your data space directly into a kernel, without first extracting features.

For instance for an email classifier, you don't need to extract word frequencies, as we've done so far, you can just design a kernel that operates directly on strings (usually based on the edit distance). If you're classifying graphs, you can design a kernel based on the Weisfeiler-Lehman graph distance metric.

using kernel SVMs

Normalize your data.

Pick a kernel (`linear`, `rbf`, `poly`)

Pick a C and the hyperparameters for your kernel (d, γ)

```
In [106]: from sklearn.svm import SVC

          lin = SVC(kernel='rbf', gamma=0.1, C=10)
          lin.fit(x, y)
```

115

why did neural networks come back?

Quadratic vs. linear training time.
SVM training needs to see all pairs of instances: $O(N^2)$
Neural net training needs k passes over the data: $O(kN)$

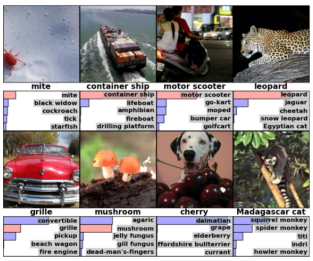
Good SVM performance required hand-designed kernels.

Deep neural nets matured, and hardware caught up with them.
LSTMs and ConvNets, both invented in 1998, provided the first breakthroughs.

Machine learning culture changed: empirical evidence of performance became acceptable without proof of convergence/learnability.

116

Neural nets require a lot of passes over the data, so it takes a big dataset before kN becomes smaller than N_2, but eventually, we got there. At that point, it became more efficient to train models by gradient descent, and the kernel trick lost its luster.



117

And when neural networks did come back, they caused a revolution. That's where we'll pick things up next lecture.

mlcourse@peterbloem.nl